RK3568 Linux的I2C总线驱动分析

2022-08-15 11:59:02 +0800

# 1. RK3568 I2C总线驱动概述

Linux的I2C接口围绕两种驱动和两种设备组织。适配器驱动(“Adapter Driver”, 总线驱动)用来抽象硬件上的I2C控制器，它挂载到其他总线上(PIC总线，platform总线)。它负责为系统中每一条I2C总线实现相应的读写方法，但本身并不进行任何通信，只是将通信方法提供给设备驱动使用，使得设备驱动可以忽略不同的I2C总线控制器的差异。

* 文件路径
  + .../OpenHarmony/out/kernel/src\_tmp/linux-5.10/drivers/i2c/busses/i2c-rk3x.c
* 设备树文件
  + .../OpenHarmony/out/kernel/src\_tmp/linux-5.10/arch/arm64/boot/dts/rockchip/rk3568.dtsi
* 总线驱动：rk3x\_i2c\_driver：
  + probe方法：rk3x\_i2c\_probe
  + remove方法：rk3x\_i2c\_remove
  + 设备树匹配：rk3x\_i2c\_match
* 总线控制器：i2c\_adapter
  + 注册i2c\_adapter：i2c\_add\_adapter
* 通信方法集合：rk3x\_i2c\_algorithm
  + 通信方法：rk3x\_i2c\_xfer
  + 功能检测方法：rk3x\_i2c\_func
* 驱动注册和卸载方法
  + module\_platform\_driver

它们的调用关系如下图：

# 2. 总线驱动定义

因为RK平台的I2C总线控制器挂载在platform总线上，总线驱动采用platform驱动框架实现，所以填充 platform\_driver 结构体以定义I2C总线驱动。

static struct platform\_driver rk3x\_i2c\_driver = {  
  
 /\*\*  
  
 \* 当驱动注册完毕，并且匹配成功，则会自动调用该方法对硬件进行操作：  
 \* a. 注册设备号，并且注册fops(为用户提供设备标示，同时提供文件操作io接口)  
 \* b. 创建设备节点  
 \* c. 初始化硬件的各项信息，如ioremap(io地址映射虚拟地址)  
 \* d. 实现各种io功能接口  
 \*/  
 .probe = rk3x\_i2c\_probe,   
  
 .remove = rk3x\_i2c\_remove,  
 .driver = {  
 .name = "rk3x-i2c", // 用于驱动和设备匹配，确保驱动和设备一致，否则无法匹配成功  
 .of\_match\_table = rk3x\_i2c\_match, // 匹配表  
 .pm = &rk3x\_i2c\_pm\_ops, // dev\_pm\_ops类型的结构体，用来赋值设备完成运行时的电源管理  
 },  
  
};

# 3. 总线控制器定义

/\*  
 \* i2c\_adapter is the structure used to identify a physical i2c bus along  
 \* with the access algorithms necessary to access it.  
 \*/  
struct i2c\_adapter {  
 struct module \*owner;  
 unsigned int class; /\* classes to allow probing for \*/  
 const struct i2c\_algorithm \*algo; /\* the algorithm to access the bus \*/  
 void \*algo\_data; /\* i2c\_algorithm的私有数据 \*/  
  
 /\* data fields that are valid for all devices, 同步机制 \*/  
 const struct i2c\_lock\_operations \*lock\_ops;  
 struct rt\_mutex bus\_lock;  
 struct rt\_mutex mux\_lock;  
  
 int timeout; /\* in jiffies \*/  
 int retries;  
 struct device dev; /\* the adapter device \*/  
 unsigned long locked\_flags; /\* owned by the I2C core \*/  
#define I2C\_ALF\_IS\_SUSPENDED 0  
#define I2C\_ALF\_SUSPEND\_REPORTED 1  
  
 int nr; /\* 适配器编号，在创建i2c\_client的时候会根据编号分类，若置为-1，则代表动态分配 \*/  
 char name[48]; /\* 适配器的名字 \*/  
 struct completion dev\_released;  
  
 struct mutex userspace\_clients\_lock;  
 struct list\_head userspace\_clients;  
  
 struct i2c\_bus\_recovery\_info \*bus\_recovery\_info;  
 const struct i2c\_adapter\_quirks \*quirks;  
  
 struct irq\_domain \*host\_notify\_domain;  
};

# 4. algorithm通信方法定义

因为每家芯片厂商SoC内部的I2C控制器是不一样的，所以 i2c\_algorithm 中直接涉及硬件层面上的代码都是由芯片商提供。例如：对I2C控制器的寄存器操作。

static const struct i2c\_algorithm rk3x\_i2c\_algorithm = {  
  
 .master\_xfer = rk3x\_i2c\_xfer, // 通信方法  
 .master\_xfer\_atomic = rk3x\_i2c\_xfer\_polling, // 通信方法，仅使用于原子上下文  
 .functionality = rk3x\_i2c\_func, // 检测通信方法支持的功能或协议，设备驱动一般会调用这个回调来确认适配器支持的协议类型  
  
};

# 5. 设备树的匹配方法

RK3568共有6个I2C控制器，分别为I2C0~I2C5，每个控制器对应不同的寄存器基地址(i2c0是0xfdd40000)， .compatible 属性都是 rockchip, rk3399-i2c ，即对应了同一个adapter驱动。

i2c0: i2c@fdd40000 {  
 compatible = "rockchip,rk3399-i2c";  
 reg = <0x0 0xfdd40000 0x0 0x1000>;  
 clocks = <&pmucru CLK\_I2C0>, <&pmucru PCLK\_I2C0>;  
 clock-names = "i2c", "pclk";  
 interrupts = <GIC\_SPI 46 IRQ\_TYPE\_LEVEL\_HIGH>;  
 pinctrl-names = "default";  
 pinctrl-0 = <&i2c0\_xfer>;  
 #address-cells = <1>;  
 #size-cells = <0>;  
 status = "disabled";  
 };  
 i2c1: i2c@fe5a0000 {  
 compatible = "rockchip,rk3399-i2c";  
 reg = <0x0 0xfe5a0000 0x0 0x1000>;  
 clocks = <&cru CLK\_I2C1>, <&cru PCLK\_I2C1>;  
 clock-names = "i2c", "pclk";  
 interrupts = <GIC\_SPI 47 IRQ\_TYPE\_LEVEL\_HIGH>;  
 pinctrl-names = "default";  
 pinctrl-0 = <&i2c1\_xfer>;  
 #address-cells = <1>;  
 #size-cells = <0>;  
 status = "disabled";  
 };  
 i2c2: i2c@fe5b0000 {  
 compatible = "rockchip,rk3399-i2c";  
 reg = <0x0 0xfe5b0000 0x0 0x1000>;  
 clocks = <&cru CLK\_I2C2>, <&cru PCLK\_I2C2>;  
 clock-names = "i2c", "pclk";  
 interrupts = <GIC\_SPI 48 IRQ\_TYPE\_LEVEL\_HIGH>;  
 pinctrl-names = "default";  
 pinctrl-0 = <&i2c2m0\_xfer>;  
 #address-cells = <1>;  
 #size-cells = <0>;  
 status = "disabled";  
 };  
 i2c3: i2c@fe5c0000 {  
 compatible = "rockchip,rk3399-i2c";  
 reg = <0x0 0xfe5c0000 0x0 0x1000>;  
 clocks = <&cru CLK\_I2C3>, <&cru PCLK\_I2C3>;  
 clock-names = "i2c", "pclk";  
 interrupts = <GIC\_SPI 49 IRQ\_TYPE\_LEVEL\_HIGH>;  
 pinctrl-names = "default";  
 pinctrl-0 = <&i2c3m0\_xfer>;  
 #address-cells = <1>;  
 #size-cells = <0>;  
 status = "disabled";  
 };  
 i2c4: i2c@fe5d0000 {  
 compatible = "rockchip,rk3399-i2c";  
 reg = <0x0 0xfe5d0000 0x0 0x1000>;  
 clocks = <&cru CLK\_I2C4>, <&cru PCLK\_I2C4>;  
 clock-names = "i2c", "pclk";  
 interrupts = <GIC\_SPI 50 IRQ\_TYPE\_LEVEL\_HIGH>;  
 pinctrl-names = "default";  
 pinctrl-0 = <&i2c4m0\_xfer>;  
 #address-cells = <1>;  
 #size-cells = <0>;  
 status = "disabled";  
 };  
 i2c5: i2c@fe5e0000 {  
 compatible = "rockchip,rk3399-i2c";  
 reg = <0x0 0xfe5e0000 0x0 0x1000>;  
 clocks = <&cru CLK\_I2C5>, <&cru PCLK\_I2C5>;  
 clock-names = "i2c", "pclk";  
 interrupts = <GIC\_SPI 51 IRQ\_TYPE\_LEVEL\_HIGH>;  
 pinctrl-names = "default";  
 pinctrl-0 = <&i2c5m0\_xfer>;  
 #address-cells = <1>;  
 #size-cells = <0>;  
 status = "disabled";  
 };

将 rk3x\_i2c\_driver 作为一个 platform\_driver 注册到内核， rk3x\_i2c\_match 作为结构体中的参数实际是一个记录设备信息的结构体数组。系统启动后会比较DTS中的匹配属性与驱动匹配表中是否相符，如果相符则会进入 rk3x\_i2c\_probe 接口，完成驱动各项参数的初始化。

// rk3x\_i2c\_driver结构体中.driver的匹配表参数  
static const struct of\_device\_id rk3x\_i2c\_match[] = {  
  
 {  
 .compatible = "rockchip,rv1108-i2c",  
 .data = &rv1108\_soc\_data  
 },  
 {  
 .compatible = "rockchip,rv1126-i2c",  
 .data = &rv1126\_soc\_data  
 },  
 {  
 .compatible = "rockchip,rk3066-i2c",  
 .data = &rk3066\_soc\_data  
 },  
 {  
 .compatible = "rockchip,rk3188-i2c",  
 .data = &rk3188\_soc\_data  
 },  
 {  
 .compatible = "rockchip,rk3228-i2c",  
 .data = &rk3228\_soc\_data  
 },  
 {  
 .compatible = "rockchip,rk3288-i2c",  
 .data = &rk3288\_soc\_data  
 },  
 {  
 .compatible = "rockchip,rk3399-i2c", // 符合DTS中描述的匹配字符串  
 .data = &rk3399\_soc\_data  
 },  
 {},  
  
};   
/\*\*  
 \* @param usb 设备名  
 \* @param skel\_table 该设备加入到模块中时对应产生的设备搜索符号  
 \* @return 生成一个名为\_\_mod\_pci\_device\_table局部变量，这个变量指向第二个参数  
 \*/  
MODULE\_DEVICE\_TABLE(of, rk3x\_i2c\_match); // 两个功能，一是将设备加入到外设队列中，二是告诉程序阅读者该设备是热插拔设备或支持热插拔功能。

# 6. 总线驱动注册和卸载方法

为了提高代码的重用性，消除多余的样板文件。当module\_init和module\_exit都不做任何特殊操作时，调用宏定义函数 module\_platform\_driver 替换 module\_init 和 module\_exit (实际还要调用一次宏定义函数 module\_driver 才能完成替换)。

/\*\*  
 \* module\_platform\_driver() - Helper macro for drivers that don't do  
 \* anything special in module init/exit.   
 \* This eliminates a lot of boilerplate.   
 \* Each module may only use this macro once, and  
 \* calling it replaces module\_init() and module\_exit()  
 \*/  
#define module\_platform\_driver(\_\_platform\_driver) \  
  
 module\_driver(\_\_platform\_driver, platform\_driver\_register, \  
 platform\_driver\_unregister)  
   
  
/\*\*  
 \* module\_driver() - Helper macro for drivers that don't do anything  
 \* special in module init/exit. This eliminates a lot of boilerplate.  
 \* Each module may only use this macro once, and calling it replaces  
 \* module\_init() and module\_exit().  
 \*  
 \* @\_\_driver: driver name  
 \* @\_\_register: register function for this driver type  
 \* @\_\_unregister: unregister function for this driver type  
 \* @...: Additional arguments to be passed to \_\_register and \_\_unregister.  
 \*  
 \* Use this macro to construct bus specific macros for registering  
 \* drivers, and do not use it on its own.  
 \*/  
#define module\_driver(\_\_driver, \_\_register, \_\_unregister, ...) \  
// 注册方法，替代module\_init  
static int \_\_init \_\_driver##\_init(void) \  
{ \  
  
 return \_\_register(&(\_\_driver) , ##\_\_VA\_ARGS\_\_); \  
  
} \  
module\_init(\_\_driver##\_init); \  
// 卸载方法，替代module\_exit  
static void \_\_exit \_\_driver##\_exit(void) \  
{ \  
  
 \_\_unregister(&(\_\_driver) , ##\_\_VA\_ARGS\_\_); \  
  
} \  
module\_exit(\_\_driver##\_exit);

# 7. 总线驱动函数

## 7.1. 初始化函数probe()

platform\_driver 在注册时会遍历 platform 总线上的 platform\_device ，当条件( .compatible 属性或 id\_table )匹配(通过得分机制，得分最高的最终会匹配成功)时，便会调用 rk3x\_i2c\_probe 函数。

/\*\*  
 \* @param pdev: 即i2c\_adapter，相当于是挂载在platform总线上的platform\_device。  
 \*/  
static int rk3x\_i2c\_probe(struct platform\_device \*pdev)  
{  
  
 struct device\_node \*np = pdev->dev.of\_node;  
 const struct of\_device\_id \*match;  
 struct rk3x\_i2c \*i2c; // 声明一个rk3x\_i2c的适配器结构体，是i2c\_adapter的进一步封装，相当于面向对象中的继承  
 int ret = 0;  
 u32 value;  
 int irq;  
 unsigned long clk\_rate;  
  
 /\*\*  
  
 \* 采用devm\_kzalloc与kzalloc相比，优点在于不用考虑释放问题，由内核完成内存回收工作  
 \* devm\_kzalloc — Resource-managed kzalloc  
 \* @param pdev: 申请内存的目标设备  
 \* @param gftp: 申请内存的类型标志，标识内存分配器将要采取的行为。其中GFP\_KERNEL最常用，五内存可用时可引起休眠。  
 \* @return: 成功返回首地址，失败返回NULL  
 \*   
  
 \* 为适配器结构体申请内存，为后续实例化完成基础工作。  
 \*/  
 i2c = devm\_kzalloc(&pdev->dev, sizeof(struct rk3x\_i2c), GFP\_KERNEL);   
 if (!i2c) // 申请失败  
 return -ENOMEM;   
  
 /\*  
  
 \* 找到rk3x\_i2c\_match数组里与之匹配的那个struct of\_device\_id  
 \*  
  
 \* i2c\_adapter驱动会兼容多个RK平台(如rk3188/rk3288/rk3399等)的i2c控制器，  
 \* 各个平台有些差异，差异性的信息可以通过.data这个成员指针保存起来，用到的时候再取出来  
 \*/  
 match = of\_match\_node(rk3x\_i2c\_match, np);  
 // 取出所匹配的i2c控制器的配置信息  
 i2c->soc\_data = match->data;  
  
 /\* use common interface to get I2C timing properties \*/  
 i2c\_parse\_fw\_timings(&pdev->dev, &i2c->t, true);  
  
 // i2c\_adapter部分成员初始化  
 // 名字  
 strlcpy(i2c->adap.name, "rk3x-i2c", sizeof(i2c->adap.name));  
 // 拥有者  
 i2c->adap.owner = THIS\_MODULE;  
 // 通信方法  
 i2c->adap.algo = &rk3x\_i2c\_algorithm;  
 i2c->adap.retries = 3;  
 i2c->adap.dev.of\_node = np;  
 i2c->adap.algo\_data = i2c;  
 i2c->adap.dev.parent = &pdev->dev;  
  
 i2c->dev = &pdev->dev;  
  
 spin\_lock\_init(&i2c->lock);  
 // 初始化等待队列头部，等待在进程调度中使用  
 init\_waitqueue\_head(&i2c->wait);  
 // 通知链机制，在内核重启之前会调用回调函数rk3x\_i2c\_restart\_notify  
 i2c->i2c\_restart\_nb.notifier\_call = rk3x\_i2c\_restart\_notify;  
 i2c->i2c\_restart\_nb.priority = 128;  
 ret = register\_pre\_restart\_handler(&i2c->i2c\_restart\_nb);  
 if (ret) {  
 dev\_err(&pdev->dev, "failed to setup i2c restart handler.\n");  
 return ret;  
 }  
 // 从dts中获取设备的物理基址  
 i2c->regs = devm\_platform\_ioremap\_resource(pdev, 0);  
 if (IS\_ERR(i2c->regs))  
 return PTR\_ERR(i2c->regs);  
  
 /\*\*  
  
 \* Switch to new interface if the SoC also offers the old one.  
 \* The control bit is located in the GRF register space.  
 \* grf\_offset: offset inside the grf regmap for setting the i2c type  
 \*/  
 if (i2c->soc\_data->grf\_offset >= 0) {  
 struct regmap \*grf;   
  
 grf = syscon\_regmap\_lookup\_by\_phandle(np, "rockchip,grf");  
 if (!IS\_ERR(grf)) {  
 int bus\_nr;  
  
 /\* Try to set the I2C adapter number from dt \*/  
 bus\_nr = of\_alias\_get\_id(np, "i2c");  
 if (bus\_nr < 0) {  
 dev\_err(&pdev->dev, "rk3x-i2c needs i2cX alias");  
 return -EINVAL;  
 }  
  
 if (i2c->soc\_data == &rv1108\_soc\_data && bus\_nr == 2)  
 /\* rv1108 i2c2 set grf offset-0x408, bit-10 \*/  
 value = BIT(26) | BIT(10);  
 else if (i2c->soc\_data == &rv1126\_soc\_data &&  
 bus\_nr == 2)  
 /\* rv1126 i2c2 set pmugrf offset-0x118, bit-4 \*/  
 value = BIT(20) | BIT(4);  
 else  
 /\* rk3xxx 27+i: write mask, 11+i: value \*/  
 value = BIT(27 + bus\_nr) | BIT(11 + bus\_nr);  
  
 ret = regmap\_write(grf, i2c->soc\_data->grf\_offset,  
 value);  
 if (ret != 0) {  
 dev\_err(i2c->dev, "Could not write to GRF: %d\n",  
 ret);  
 return ret;  
 }  
 }  
 }  
  
 /\* IRQ setup(中断设置) \*/  
 irq = platform\_get\_irq(pdev, 0); // platform\_device结构体中存储有所用到的中断号  
 if (irq < 0)  
 return irq;  
  
 ret = devm\_request\_irq(&pdev->dev, irq, rk3x\_i2c\_irq,  
 0, dev\_name(&pdev->dev), i2c);  
 if (ret < 0) {  
 dev\_err(&pdev->dev, "cannot request IRQ\n");  
 return ret;  
 }  
  
 // 存储用户主动申请的内存区域指针防止丢失  
 platform\_set\_drvdata(pdev, i2c);  
  
 // 平台时钟设置  
 if (i2c->soc\_data->calc\_timings == rk3x\_i2c\_v0\_calc\_timings) {  
 /\* Only one clock to use for bus clock and peripheral clock \*/  
 i2c->clk = devm\_clk\_get(&pdev->dev, NULL);  
 i2c->pclk = i2c->clk;  
 } else {  
 i2c->clk = devm\_clk\_get(&pdev->dev, "i2c");  
 i2c->pclk = devm\_clk\_get(&pdev->dev, "pclk");  
 }  
  
 if (IS\_ERR(i2c->clk))  
 return dev\_err\_probe(&pdev->dev, PTR\_ERR(i2c->clk),  
 "Can't get bus clk\n");  
  
 if (IS\_ERR(i2c->pclk))  
 return dev\_err\_probe(&pdev->dev, PTR\_ERR(i2c->pclk),  
 "Can't get periph clk\n");  
  
 ret = clk\_prepare(i2c->clk);  
 if (ret < 0) {  
 dev\_err(&pdev->dev, "Can't prepare bus clk: %d\n", ret);  
 return ret;  
 }  
 ret = clk\_prepare(i2c->pclk);  
 if (ret < 0) {  
 dev\_err(&pdev->dev, "Can't prepare periph clock: %d\n", ret);  
 goto err\_clk;  
 }  
  
 i2c->clk\_rate\_nb.notifier\_call = rk3x\_i2c\_clk\_notifier\_cb;  
 ret = clk\_notifier\_register(i2c->clk, &i2c->clk\_rate\_nb);  
 if (ret != 0) {  
 dev\_err(&pdev->dev, "Unable to register clock notifier\n");  
 goto err\_pclk;  
 }  
  
 clk\_rate = clk\_get\_rate(i2c->clk);  
 rk3x\_i2c\_adapt\_div(i2c, clk\_rate);  
  
 // 向内核添加i2c\_adapter  
 ret = i2c\_add\_adapter(&i2c->adap);  
  
 if (ret < 0)  
 goto err\_clk\_notifier;  
  
 return 0;  
  
err\_clk\_notifier:  
  
 clk\_notifier\_unregister(i2c->clk, &i2c->clk\_rate\_nb);  
  
err\_pclk:  
  
 clk\_unprepare(i2c->pclk);  
  
err\_clk:  
  
 clk\_unprepare(i2c->clk);  
 return ret;  
  
}

## 7.2. 释放函数remove()

static int rk3x\_i2c\_remove(struct platform\_device \*pdev)  
{  
  
 // 获得rk3x\_i2c\_probe中存储的主动申请的内存区域指针  
 struct rk3x\_i2c \*i2c = platform\_get\_drvdata(pdev);  
 // 卸载i2c\_adapter  
 i2c\_del\_adapter(&i2c->adap);  
  
 // 注销时钟  
 clk\_notifier\_unregister(i2c->clk, &i2c->clk\_rate\_nb);  
 unregister\_pre\_restart\_handler(&i2c->i2c\_restart\_nb);   
 clk\_unprepare(i2c->pclk);  
 clk\_unprepare(i2c->clk);   
  
 return 0;  
  
}

## 7.3. 通信方法函数master\_xfer()

实现总线上数据传输，rk3x\_i2c提供了两种方式来处理阻塞进程唤醒。

// 使用等待队列实现阻塞进程唤醒  
static int rk3x\_i2c\_xfer(struct i2c\_adapter \*adap,   
  
 struct i2c\_msg \*msgs, int num)  
  
{  
  
 return rk3x\_i2c\_xfer\_common(adap, msgs, num, false);  
  
}  
// 使用poll机制而非等待队列实现阻塞进程唤醒  
static int rk3x\_i2c\_xfer\_polling(struct i2c\_adapter \*adap,   
  
 struct i2c\_msg \*msgs, int num)  
  
{  
  
 return rk3x\_i2c\_xfer\_common(adap, msgs, num, true);  
  
}  
static int rk3x\_i2c\_xfer\_common(struct i2c\_adapter \*adap,   
  
 struct i2c\_msg \*msgs, int num, bool polling)  
  
{  
  
 struct rk3x\_i2c \*i2c = (struct rk3x\_i2c \*)adap->algo\_data;  
 unsigned long timeout, flags;  
 u32 val;  
 int ret = 0;  
 int i;  
  
 if (i2c->suspended)  
 return -EACCES;  
  
 spin\_lock\_irqsave(&i2c->lock, flags);  
 // 使能时钟  
 clk\_enable(i2c->clk);  
 clk\_enable(i2c->pclk);  
  
 i2c->is\_last\_msg = false;  
  
 /\*  
  
 \* Process msgs. We can handle more than one message at once (see  
 \* rk3x\_i2c\_setup()).  
 \* i+ret就是指一次循环处理多条msg  
 \*/  
 for (i = 0; i < num; i += ret) {  
 ret = rk3x\_i2c\_setup(i2c, msgs + i, num - i);  
  
 // i2c msg处理出错  
 if (ret < 0) {  
 dev\_err(i2c->dev, "rk3x\_i2c\_setup() failed\n");  
 break;  
 }  
 // ret为本轮已处理的msgs数量，i为过去已处理的msgs数量，如果刚好等于msgs总数，则  
 if (i + ret >= num)  
 i2c->is\_last\_msg = true;  
 // 启动i2c\_adapter  
 rk3x\_i2c\_start(i2c);   
 // 释放自旋锁，并恢复标志寄存器的值为变量flags保存的值  
 spin\_unlock\_irqrestore(&i2c->lock, flags);  
  
 // 区分调用的是rk3x\_i2c\_xfer还是rk3x\_i2c\_xfer\_polling  
 if (!polling) {  
 /\*\*  
  
 \* 调用rk3x\_i2c\_xfer, 挂起等待队列头i2c->wait(wait\_queue\_head\_t)。  
 \* 直到i2c\_adapter不再繁忙或者等待超时，则调用wake\_up(i2c->wait)会唤醒进程并继续往下执行  
 \* 如果未超时且i2c\_adapter繁忙，则进程会被阻塞  
 \*/  
  
 timeout = wait\_event\_timeout(i2c->wait, !i2c->busy,  
 msecs\_to\_jiffies(WAIT\_TIMEOUT));  
 } else {  
 timeout = rk3x\_i2c\_wait\_xfer\_poll(i2c);  
 }  
 // 获得自旋锁，并且备份本地中断和中断状态到flags变量  
 spin\_lock\_irqsave(&i2c->lock, flags);  
  
 // 超时了  
 if (timeout == 0) {  
 dev\_err(i2c->dev, "timeout, ipd: 0x%02x, state: %d\n",  
 i2c\_readl(i2c, REG\_IPD), i2c->state);  
  
 /\* Force a STOP condition without interrupt \*/  
 rk3x\_i2c\_disable\_irq(i2c);  
 val = i2c\_readl(i2c, REG\_CON) & REG\_CON\_TUNING\_MASK;  
 val |= REG\_CON\_EN | REG\_CON\_STOP;  
 i2c\_writel(i2c, val, REG\_CON);  
  
 i2c->state = STATE\_IDLE;  
  
 ret = -ETIMEDOUT;  
 break;  
 }  
  
 // 未超时，但是i2c\_adapter运行中产生错误  
 if (i2c->error) {  
 ret = i2c->error;  
 break;  
 }  
 }  
  
 // 顺利完成i2c通信？  
 // 关闭中断  
 rk3x\_i2c\_disable\_irq(i2c);  
 // 关闭i2c总线  
 rk3x\_i2c\_disable(i2c);  
 // 关闭时钟  
 clk\_disable(i2c->pclk);  
 clk\_disable(i2c->clk);  
  
 spin\_unlock\_irqrestore(&i2c->lock, flags);  
  
 // 出错返回ret，成功返回处理msgs的数量  
 return ret < 0 ? ret : num;  
  
}  
  
// 使用poll机制而非等待队列  
static int rk3x\_i2c\_wait\_xfer\_poll(struct rk3x\_i2c \*i2c)  
{  
  
 // ktime\_get()得到当前时间，ktime\_add\_ms来设置超时时间。  
 ktime\_t timeout = ktime\_add\_ms(ktime\_get(), WAIT\_TIMEOUT);  
  
 /\*\*  
  
 \* READ\_ONCE是一个用来读取变量的宏，为了避免编译器优化导致多线程时读取出错而编写  
 \* ktime\_compare用来确定是否超时  
 \*/  
 while (READ\_ONCE(i2c->busy) &&  
 ktime\_compare(ktime\_get(), timeout) < 0) {  
 // 没超时且i2c\_adapter繁忙，则延时等待  
 udelay(5);   
 // 设置中断  
 rk3x\_i2c\_irq(0, i2c);   
 }  
 // 返回i2c\_adapter的状态  
 return !i2c->busy;   
  
}

## 7.4. 功能检测函数func()

// 无格式i2c-level命令(Pure SMBus适配器不能用这些命令)  
#define I2C\_FUNC\_I2C 0x00000001  
// 处理所有的能够被I2C adapter仿真的SMBus命令  
#define I2C\_FUNC\_SMBUS\_EMUL (I2C\_FUNC\_SMBUS\_QUICK | \  
  
 I2C\_FUNC\_SMBUS\_BYTE | \  
 I2C\_FUNC\_SMBUS\_BYTE\_DATA | \  
 I2C\_FUNC\_SMBUS\_WORD\_DATA | \  
 I2C\_FUNC\_SMBUS\_PROC\_CALL | \  
 I2C\_FUNC\_SMBUS\_WRITE\_BLOCK\_DATA | \  
 I2C\_FUNC\_SMBUS\_I2C\_BLOCK | \  
 I2C\_FUNC\_SMBUS\_PEC)  
  
// 熟知的有I2C\_M\_IGNORE\_NAK, I2C\_M\_REV\_DIR\_ADDR, I2C\_M\_NOSTART, I2C\_MNO\_RD\_ACK等flags(I2C寄存器状态？)  
#define I2C\_FUNC\_PROTOCOL\_MANGLING 0x00000004 /\* I2C\_M\_IGNORE\_NAK etc. \*/  
// 处理SMBus write\_quick命令  
#define I2C\_FUNC\_SMBUS\_QUICK 0x00010000  
// 处理SMBus read\_byte & write\_byte命令  
#define I2C\_FUNC\_SMBUS\_BYTE (I2C\_FUNC\_SMBUS\_READ\_BYTE | \  
  
 I2C\_FUNC\_SMBUS\_WRITE\_BYTE)  
  
// 处理SMBus read\_byte\_data & write\_byte\_data命令  
#define I2C\_FUNC\_SMBUS\_BYTE\_DATA (I2C\_FUNC\_SMBUS\_READ\_BYTE\_DATA | \  
  
 I2C\_FUNC\_SMBUS\_WRITE\_BYTE\_DATA)  
  
// 处理SMBus read\_word\_data & write\_word\_data命令  
#define I2C\_FUNC\_SMBUS\_WORD\_DATA (I2C\_FUNC\_SMBUS\_READ\_WORD\_DATA | \  
  
 I2C\_FUNC\_SMBUS\_WRITE\_WORD\_DATA)  
  
// 处理SMBus process\_call命令  
#define I2C\_FUNC\_SMBUS\_PROC\_CALL 0x00800000  
// 处理SMBus wrtie\_block\_data命令  
#define I2C\_FUNC\_SMBUS\_WRITE\_BLOCK\_DATA 0x02000000  
// 处理SMBus read\_i2c\_block\_data & write\_i2c\_block\_data命令  
#define I2C\_FUNC\_SMBUS\_I2C\_BLOCK (I2C\_FUNC\_SMBUS\_READ\_I2C\_BLOCK | \  
  
 I2C\_FUNC\_SMBUS\_WRITE\_I2C\_BLOCK)  
  
#define I2C\_FUNC\_SMBUS\_PEC 0x00000008  
// 处理SMBus read\_byte命令  
#define I2C\_FUNC\_SMBUS\_READ\_BYTE 0x00020000  
// 处理SMBus write\_byte命令  
#define I2C\_FUNC\_SMBUS\_WRITE\_BYTE 0x00040000  
// 处理SMBus read\_byte\_data命令  
#define I2C\_FUNC\_SMBUS\_READ\_BYTE\_DATA 0x00080000  
// 处理SMBus write\_byte\_data命令  
#define I2C\_FUNC\_SMBUS\_WRITE\_BYTE\_DATA 0x00100000  
// 处理SMBus read\_word\_data命令  
#define I2C\_FUNC\_SMBUS\_READ\_WORD\_DATA 0x00200000  
// 处理SMBus write\_word\_data命令  
#define I2C\_FUNC\_SMBUS\_WRITE\_WORD\_DATA 0x00400000  
// 处理SMBus read\_i2c\_block\_data命令  
#define I2C\_FUNC\_SMBUS\_READ\_I2C\_BLOCK 0x04000000 /\* I2C-like block xfer \*/  
// 处理SMBus write\_i2c\_block\_data命令  
#define I2C\_FUNC\_SMBUS\_WRITE\_I2C\_BLOCK 0x08000000 /\* w/ 1-byte reg. addr. \*/  
  
// 列出所支持的命令集(检测通信方法支持的功能或协议)  
static u32 rk3x\_i2c\_func(struct i2c\_adapter \*adap)  
{  
  
 return I2C\_FUNC\_I2C | I2C\_FUNC\_SMBUS\_EMUL | I2C\_FUNC\_PROTOCOL\_MANGLING;  
  
}

## 7.5. i2c\_adapter注册函数i2c\_add\_adapter()